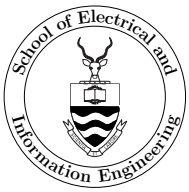

UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Electrical and Information Engineering
Software Development II

A Brief Guide To Writing Unit Tests

1 Create separate projects for test builds and production builds

Do not mix test code with production code. Practically, this means creating two different projects in your IDE which *share the common source files* containing the classes being tested. One project will also contain all of your tests and include the `googletest` framework, while the other will contain the `main` function of your actual program and additional source files that are not being tested (such as those dealing with the user interface). Note, do not make two copies of the source code being tested — this violates the DRY principle!

2 Choose a good test name

It is important to name tests well. A good name is quite specific in describing the *behaviour* being tested, the expected result, and possibly the motivation for the test.

For example, the following test names for the `Word` class in Laboratory 3 are poor:

```
TEST(Word, Constructor) // test named after member function
TEST(Word, onCreationFormatsWord) // lacks detail about what formatting entails
```

Rather use:

```
TEST(Word, onCreationRemovesPunctuation)
```

Remember that test names can be as long as you like, as you only need to type them out once — the test framework is responsible for actually calling the test.

3 Keep test code simple

The motivation for this is that you want to minimise the possibility of introducing errors into your test code. To keep the code simple use the following guidelines:

- Each unit test should only verify a *single* behaviour or responsibility of the object. This does not mean that a unit test must only call a single member function of the object, as more than one member function might be needed to verify the behaviour. This also does mean that your unit test must only contain a single assertion, as more than one assertion might be necessary. However, if you have a lot of different member function calls or assertions in one test, you may need to split the test up into more tests at a finer level of granularity.
- Avoid conditional code in your tests (`if` statements etc). You can remove an `if` statement, for example, by having two separate tests, one for each branch of the conditional.

4 Do not expose private data in order to write a test

Tests should be written against the public interface of a class, just as production code will be. You should not add special member functions (such as 'get' functions) to your class in order to enable testing. This makes your test code brittle as it becomes coupled to the object's implementation.

Test private member functions through the public member functions which use them.

5 Test-first or test-last?

Many developers are of the opinion that writing the tests before the code is beneficial (test-driven development). You may or may not like this approach, and there is no requirement in this course that you write your tests first (or last). The important point is keep the *time gap* between writing a test, and writing the code that makes the test pass, *small*. If you do this you establish a tight feedback loop between writing code and verifying its behaviour. So, if you write a test last, write it immediately after writing the function which needs to be tested. Also, do not create very high-level tests which will require hours of work before you will be in a position to make the tests pass.

Version 1.01 (28/7/2016)
`guide-to-writing-unit-tests.tex`